12.1.2 Array-Based Implementation of Merge-Sort

We begin by focusing on the case when a sequence of items is represented with an array. The merge method (Code Fragment 12.1) is responsible for the subtask of merging two previously sorted sequences, S_1 and S_2, with the output copied into S. We copy one element during each pass of the while loop, conditionally determining whether the next element should be taken from S_1 or S_2. The divide-and-conquer merge-sort algorithm is given in Code Fragment 12.2.

We illustrate a step of the merge process in Figure 12.5. During the process, index i represents the number of elements of S_1 that have been copied to S, while index j represents the number of elements of S_2 that have been copied to S. Assuming S_1 and S_2 both have at least one uncopied element, we copy the smaller of the two elements being considered. Since $i+j$ objects have been previously copied, the next element is placed in $S[i+j]$. (For example, when $i+j$ is 0, the next element is copied to $S[0]$). If we reach the end of one of the sequences, we must copy the next element from the other.

```
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];                  // copy ith element of S1 and increment i
    else
      S[i+j] = S2[j++];                  // copy jth element of S2 and increment j
  }
}
```

Code Fragment 12.1: An implementation of the merge operation for a Java array.

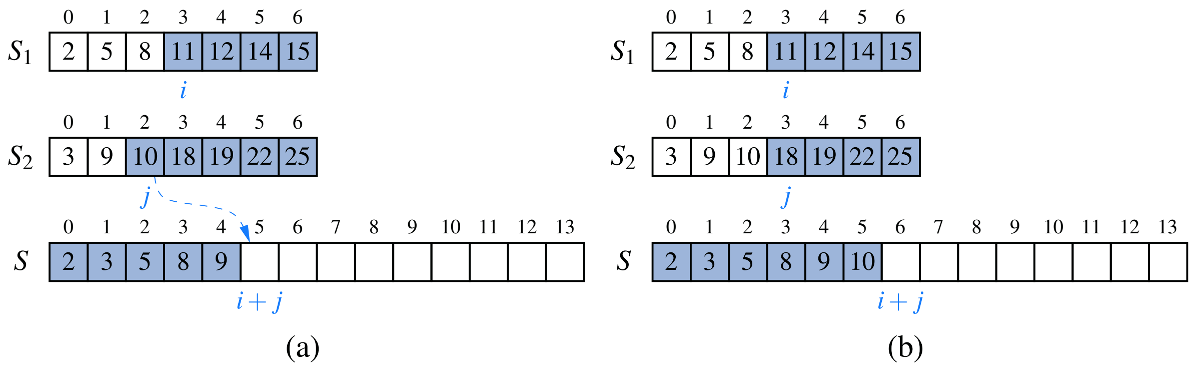

Figure 12.5: A step in the merge of two sorted arrays for which $S_2[j] < S_1[i]$. We show the arrays before the copy step in (a) and after it in (b).

```
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                              // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);        // copy of first half
  K[ ] S2 = Arrays.copyOfRange(S, mid, n);        // copy of second half
  // conquer (with recursion)
  mergeSort(S1, comp);                            // sort copy of first half
  mergeSort(S2, comp);                            // sort copy of second half
  // merge results
  merge(S1, S2, S, comp);                // merge sorted halves back into original
}
```

Code Fragment 12.2: An implementation of the recursive merge-sort algorithm for a Java array (using the merge method defined in Code Fragment 12.1).

We note that methods merge and mergeSort rely on use of a Comparator instance to compare a pair of generic objects that are presumed to belong to a total order. This is the same approach we introduced when defining priority queues in Section 9.2.2, and when studying implementing sorted maps in Chapters 10 and 11.

12.1.3 The Running Time of Merge-Sort

We begin by analyzing the running time of the merge algorithm. Let n_1 and n_2 be the number of elements of S_1 and S_2, respectively. It is clear that the operations performed inside each pass of the while loop take $O(1)$ time. The key observation is that during each iteration of the loop, one element is copied from either S_1 or S_2 into S (and that element is considered no further). Therefore, the number of iterations of the loop is $n_1 + n_2$. Thus, the running time of algorithm merge is $O(n_1 + n_2)$.

Having analyzed the running time of the merge algorithm used to combine subproblems, let us analyze the running time of the entire merge-sort algorithm, assuming it is given an input sequence of n elements. For simplicity, we restrict our attention to the case where n is a power of 2. We leave it to an exercise (R-12.3) to show that the result of our analysis also holds when n is not a power of 2.

When evaluating the merge-sort recursion, we rely on the analysis technique introduced in Section 5.2. We account for the amount of time spent within each recursive call, but excluding any time spent waiting for successive recursive calls to terminate. In the case of our mergeSort method, we account for the time to divide the sequence into two subsequences, and the call to merge to combine the two sorted sequences, but we exclude the two recursive calls to mergeSort.